December 2010

Analyzing Trust by Scripting CPSA

Benjamin Erik Petrin
Worcester Polytechnic Institute

Analyzing Trust by Scripting CPSA

A Major Qualifying Project Report
submitted to the faculty of

Worcester Polytechnic Institute

in partial fulfillment of the requirements
for the degree of Bachelor of Science

in

Computer Science

by

Benjamin Petrin

December 2010

APPROVED:

Professor Joshua Guttman, project advisor

Professor Kathi Fisler, project co-advisor

Abstract

This paper describes algorithms for use with the Cryptographic Protocol Shape Analyzer to automate and simplify tasks relating to cryptographic protocol analysis. A Python implementation of these algorithms is provided and documented. The tools developed in this project will be useful to both protocol designers and students looking to gain an understanding in trust engineering.

Table of Contents

1 Introduction

Cryptographic protocol design is an error-prone process even under the assumptions of an ideal cryptographic system. For this reason, a method of describing cryptographic protocols and proving their correctness is of great utility. The Cryptographic Protocol Programming Language (CPPL) developed by the Mitre Corporation provides the syntax and semantics necessary to describe nonce-based cryptographic protocols in a manner that is conducive to compilation [3]. Additionally, the Cryptographic Protocol Shape Analyzer (CPSA) was developed to allow protocol designers to analyze cryptographic protocols and prove the protocols fulfill specific security goals.

CPSA allows for the analysis of distributed cryptographic protocols in which participants in the protocol may trust other participants for some information but not necessarily all information. Distributed systems based on these cryptographic protocols often need to coordinate state changes of multiple participants, with each participant possessing knowledge only of what it has sent and received. This notion of analyzing protocol execution based on the local behavior of single participants allows for security theorems to be observed while accounting for possible adversarial behavior attempting to interrupt business-as-usual communication.

1.1 Goals of this paper

CPSA allows the protocol designer to observe or "read off" proof of a cryptographic protocol's behavior. For instance, a protocol designer may use CPSA in conjunction with a particular protocol definition to observe that an adversary cannot gain access to a particular value being transmitted during the execution of the protocol. Additionally, the protocol designer can annotate protocols with trust information to indicate what each participant is promising by sending a message and what each participant learns by receiving a message. This project was an attempt to improve the usefulness of CPSA with respect to what information may be "read off" of a cryptographic protocol that is decorated with trust information after some processing. The protocol designer can use the tools produced by this project in conjunction with CPSA in order to learn new information about the protocol she is designing. The information revealed is useful to potentially two different entities; the human designing the protocol and the protocol participants themselves.

First, the human who is annotating a protocol stands to gain insight into the interplay

of trust information and protocol execution. Instead of a protocol designer creating a protocol that appears as if it accomplishes its goals, a protocol designer can gain an understanding in precisely how the ingredients of a message affect each goal. Such computer aided discovery reduces the effort required in producing sound protocols of increasingly more complex systems. In a hypothesis-test-refine workflow, protocol designers can move confidentially toward achieving their security goals while a system verifies their intuition.

Similarly, the information gleaned from an analysis of a protocol should specify exactly what information may be used by a particular participant in the protocol at different points of a protocol execution. This run-time information can serve as the input to the underlying software implementation of the protocol. The information provides a rulebook describing the preconditions necessary to continue with a protocol's execution, branch, or stop protocol execution altogether.

This project aims to decrease where trust information needs to be added to a protocol by the protocol designer. Additionally, it seeks to reflect security constraints in a manner congruent with the execution of a protocol in order to allow the protocol designer to explore protocol execution in a meaningful way. These goals are further motivated in Section 4 after relevant background knowledge has been introduced in Section 2.

1.2 How this paper is organized

This paper begins by by introducing relevant background knowledge in order to explain the aforementioned shortcomings of the current use of the CPPL and the corresponding analysis tool CPSA. This paper motivates these problems further before describing the solutions derived as part of this project.

The implications and usefulness of each of the solutions provided by this project are analyzed and discussed. The paper concludes with future work to be done and new questions that have been discovered as a result of this project's execution.

1.3 What was learned

In working with a distributed system such as a distributed cryptographic protocol, knowing what can and cannot be said to be *true* of the participants in the system is a very powerful notion. Embodied as we are, evaluating a system that takes place in multiple places at once possess a challenge unlike that of any system based on a single point of execution.

Such single-point applications can be related to more easily in much the same way we can imagine following a cake recipe as a means of evaluating whether it will produce the desired result or if the recipe only appears on the surface to accomplish its goals. How do we go about evaluating if a cryptographic protocol will produce some desired result such as fulfil a confidentiality goal when all we have is the raw messages exchanged by each participant? As the complexity of a distributed system increases, so too do the demands on the human working with the system who tries to evaluate how and why a distributed system behaves the way it does. Any effort to simplify this task without adversely sacrificing expressiveness is useful.

In working with this project I have learned that CPSA provides an intuitive way to go about analyzing a distributed system. By fixing itself on the point of a single participant at a particular time, the model used by CPSA has advantages in relating to our own view of each participant as an individual "black box". If we allow ourselves to personify each participant in the protocol: exploring protocol behavior from a single participant's local view maps well on to our way of exploring the world; we take input from our immediate surroundings and attempt to conclude on what else must be true in the rest of the environment solely from these limited inputs. As I worked with CPSA, I found the model it uses to represent protocol behavior is one very compatible with my way of exploring problems.

Although the idea of exploring protocols from the view of individual participants arose out of the mathematical assumptions of trust underlying the problem, this nicety in how it represented problems from single points was one I found particularly helpful in understanding how and why protocols behave in a particular way. As we might expect from any well designed system, CPSA provides the abstractions necessary to allow a complicated problem to be modeled in a simpler way.

The solutions presented in this paper are logical extensions of this model. The theme of exploring protocol behavior is emphasized continuously as I have found it to be the way in which I learn the most about how a protocol works. This observation has led me to believe that not only will CPSA and the tools presented in this paper be of use to protocol designers, they can also be useful in teaching protocol behavior and design patterns to students. By exploring how ingredients of a protocol effect its execution, a student can gain insight into why existing protocols are designed the way they are.

2 Technical Background

2.1 Cryptography Basics

Cryptography is concerned with the use of messages to securely transmit information between participants. A cryptographic protocol may have goals such as preserving the privacy of information and authenticating participants. This paper is concerned with *nonce-based* cryptography. Messages in cryptographic systems are composed of various ingredients combined and manipulated by various cryptographic operations.

The operation of *encryption* transforms a plaintext into a ciphertext by the use of some encryption key. The plaintext can be recovered from the otherwise random looking ciphertext with knowledge of the necessary and corresponding decryption key. In *symmetric key encryption*, the keys used to encrypt and decrypt a particular message are identical. Such a key is commonly referred to as a shared key since the participants wishing to use that key to transmit information in such a way that others cannot view it must each share the otherwise secret key and not expose it to adversaries [5]. In this paper and as part of the input language to CPSA, operations such as the encryption operation will often be written in prefix function notation, where a function is written before its arguments and wrapped in parenthesis such as `(enc m k)` to represent the operation of encrypting message `m` with key `k`.

In contrast to symmetric key encryption, *asymmetric key encryption* utilizes different keys for the operations of encrypting and decrypting messages. This form of encryption is sometimes referred to as *public-key encryption* as one key (the encryption key) is often advertised publicly allowing any willing would-be message transmitter to encrypt messages destined for the owner of the public key. Presumably, the owner alone possess the private key necessary to decrypt the ciphertext to recover the plainext that was encrypted using the public key [5].

Asymmetric key encryption is often mentioned along side the operation of *signing* which allows a participant to manipulate a message using her private key in a manner that can be easily checked by others with knowledge of only the signer's public key. Signing forms the basis of knowing reliably that a particular participant has said something.

The operation of *hashing* allows a participant to take a message and modify it in an irreversible but reproducible way. Conceptually, this is similar to public key encryption for which no participant possess the corresponding private key. It follows that two participants

could take a piece of plaintext and hash it to produce identical cyphertexts (hashes), but no participant could use the hash to recover the plaintext that was used as input to the hash operation.

The simplest of operations would be that of concatenation. Ingredients for a message may be combined into one either to be transmitted or used as arguments to one of the various other cryptographic operations already described. It is assumed here that participants can decouple message ingredients after receiving the result of their concatenation.

Lastly a *nonce* is a special message ingredient that can be generated by participants to serve as a sort of transaction number. Like a transaction number, nonces are freshly generated on each run of a protocol. Unlike transaction numbers which may be sequential, knowledge of a nonce generated from one run of a cryptographic protocol does not provide any way of guessing the nonce generated on another execution of the same protocol.

2.2 Real-world Cryptography

This paper addresses problems relating to protocol insecurity even assuming a perfect cryptographic system. Such a system has no underlying weaknesses in its implementation. For instance, although a private key and its corresponding public key must be related mathematically, we assume that there is no feasible way to deduce the private key from the public key. The use of prime-number based encryption schemes such as RSA provide a close approximation of these goals at present. Similarly, we assume that nonces are sufficiently large random numbers that cannot be reliably guessed. Many hashing algorithms in use today have a finite output leading to collisions or the same hash output for two different inputs. Like guessing a nonce, a good hash function will make it mathematically infeasible to calculate plaintexts to produce a given hash.

It follows that although the encryption, hashing, and signing algorithms may not be perfect, they provide (at present) the sort of reasonable expectations needed to make secure communications. CPPL, CPSA, and the problems addressed in this paper are not concerned with the closeness such implementations are to an ideal cryptographic systems but instead the weaknesses possible in even a perfect cryptographic system. By extension, these weaknesses have implications in today's less-than-perfect cryptographic protocols as well.

Protocols written against perfect cryptographic systems may still possess weaknesses due to the nature and meaning of the messages transmitted. Historically, such weaknesses

have been discovered and addressed through lengthy discussion among peers. CPSA attempts to address these weaknesses in a systematic way. CPSA allows a cryptographic protocol designer to prove security theorems in a protocol and create counterexamples for protocols with weaknesses.

2.3 CPSA and Strand Spaces

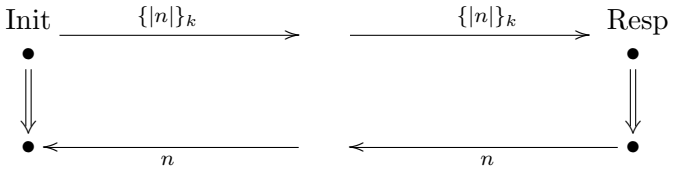
CPSA describes cryptographic protocols with respect to their structure, lending to a representation known as *strand spaces*. In this representation, each *strand* represents one trace sequence of events (both message receptions and transmissions) of a particular participant in a nonce-based cryptographic protocol. Each *node* on the strand represents only the transmission or reception of a message. The strand and node semantics make no further assumptions about the message such as where it came from or which participant should receive it.

This representation clarifies the use of the Strand Space model to account for adversarial behavior. An adversary may, for instance, observe a message in transmission and potentially decrypt the message if the adversary possess the necessary keys. Similarly, an adversary could produce messages of its own that look like those originating from well-behaved participants, providing the adversary has the knowledge necessary to do. This paper addresses the limitations and abilities of adversaries towards the end of this section.

Messages are constructed from atoms which are instantiated with values during protocol execution. Cryptographic operations such as symmetric or asymmetric key encryption, signing, or hashing (formalized here as asymmetric key encryption for which no participant possess the private decryption key) may be applied to atoms and the concatenation of atoms [3].

A particular atom is an *ingredient* of a message if it is found in the message either in or outside the application of a cryptographic function. We write, for example, $t_0 \sqsubseteq \{|t_0 \hat{} t_1|\}_k$ to mean that t_0 is an ingredient to the message consisting of the encryption of the concatenation of t_0 with t_1 using the key k. Ingredients do not include the keys so $k \not\sqsubseteq \{|t_0 \hat{} t_1|\}_k$ providing, of course, $k \not\sqsubseteq t_0$ and $k \not\sqsubseteq t_1$.

Strands are depicted horizontally or vertically with nodes as bullets and the transition between nodes on a single strand drawn with double arrows as in $\bullet \Longrightarrow \bullet$. Strands never split or join; possible branches of behavior are depicted with separate strands. If a role could receive one message on one execution of a protocol and another message of different



(a) The initiator strand (b) The responder strand

Figure 1: The regular strands of a simple example protocol

ingredients on another, two strands would be used to represent the branching behavior of this role. Note that messages with different ingredients necessitate the use of a new strand to represent the different behavior. This is not to say that a separate strand is necessary to represent all the possible values bound to a particular ingredient at runtime. Ingredients represent the nature of a piece of information carried in a message, such as a nonce or a text value, and not the actual value bound to it.

Protocols are defined by specifying the trace sequences of sends and receives for the one or more strands that are necessary to represent the behavior of each participant. Thus a protocol Π is a set of strands for well-behaved participants of a cryptographic protocol. Such strands are referred to as *regular* strands [3]. Absent from the protocol definition are *penetrator* strands, traces of adversarial participants that may attempt to interfere with the goals of a cryptographic protocol. We may address nodes on strands by their position such as with $a \downarrow 3$ to refer to the third node on strand a.

Figure 1 shows the definition of two strands that make up a simple example protocol. In this example, an initiator sends a nonce encrypted with key k and receives back the nonce unencrypted. A responder strand receives a nonce encrypted with key k and transmits the same nonce. Note that as this figure shows the definition of the protocol and not an execution of this protocol, the two strands are not connected to each other, as a protocol definition makes no assumptions as to the messages received by each participant. If we look at a particular execution of this same protocol, such as in Figure 2, we see the strands are connected to indicate the message sent from participant was received by the other. Such an execution could be possible if, for example the responder possessed the shared key k or already had knowledge of the nonce n. Figure 2 alone does not provide the knowledge necessary to distinguish between the two different preconditions that could have lead to the execution observed.

The strand space consists of all strands including legitimate strands specified as part of the protocol definition and adversarial strands whose trace sequences may take on any number of configurations. A *bundle* represents these strands attached by matching

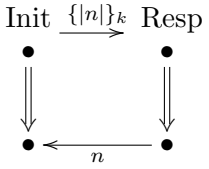


Figure 2: A simple example protocol execution

message transmissions and receptions [4].

2.4 CPSA and Shapes

The Cryptographic Protocol Shape Analyzer (CPSA) developed by the Mitre Corporation enumerates the possible *shapes* or essentially different executions of a nonce-based cryptographic protocol. As shown in Figure 2, shapes are concerned with the form of the messages exchanged and not the various possible values assigned to nonces or other variable message ingredients; as a result there are often only a small number of shapes that may result from a particular protocol definition. The small number of shapes make them useful in exploring the behavior of a protocol and determining if it fulfils particular goals such as authentication or secrecy [1].

The protocol designer queries CPSA for possible shapes that are compatible with a particular *skeleton*. A skeleton is a partial ordering of regular (non-adversarial) strands [2]. When a skeleton is used as a query to CPSA, it represents the observation of a particular node of interest. That is, a skeleton, predisposes a particular message transmission or reception, and posits the question "What other strands must have been involved for this node to have been observed?". The involved strands may either be legitimate participants in the cryptographic protocol or penetrators. All participants, however, are bound by certain assumptions regarding the origination of certain message contents. This origination information (such as which participants possess which keys) is part of the query skeleton scenario and is explored in detail shortly.

CPSA searches for the essentially different ways to justify the nodes of the skeleton. This often involves adding strands that are necessary to explain how a particular strand received a piece of information necessary to send some message. The result of this expansion is more skeletons with strands and nodes added to fully explain all messages. The strands are related by matched message transmissions and receptions between the strands. Each skeleton produced by a particular skeleton query is a *homomorphism*. The skeletons that are *realized* or result from the execution of a protocol explain all message

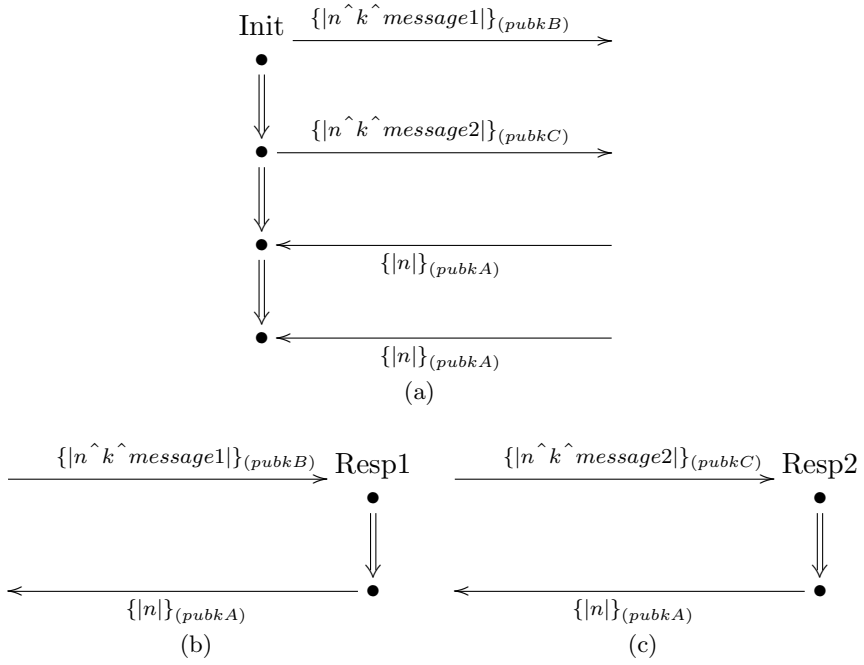
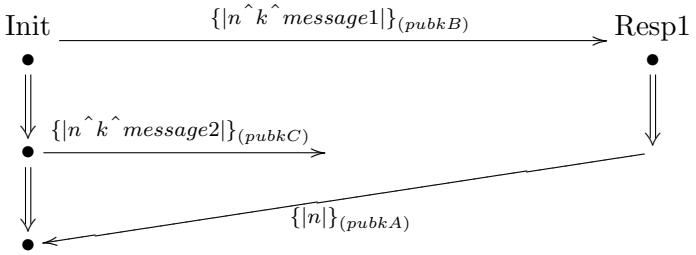


Figure 3: A simple protocol definition with multiple shapes

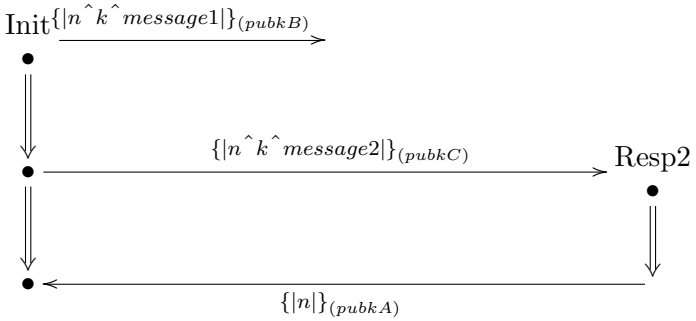
transmissions and receptions posited by the skeleton query.

As an example, consider Figure 3 which involves an initiator that transmits two distinct messages each containing the same nonce. The first message is encrypted with the public key of strand b and the second with the public key of strand c. Two separate responders receive and decrypt the nonce using their respective private keys and each responds with the resulting nonce plaintext. The misalignment of the message transmissions and receptions is intentional here to suggest that these are strand definitions making up a protocol and not a particular execution of this protocol. What can the initiator conclude about the behavior of others after observing the reception of a message on its 3rd node of strand a? If we query CPSA for shapes compatible with a skeleton observing this node operating under the assumptions of a freshly generated nonce and uncompromised private keys for b and c, two distinct shapes are generated. The two essentially different shapes represent the two possible executions that could describe the observed message reception and are shown in Figure 4.

The fact that the keys were uncompromised and the nonce was freshly generated are details we will refer to as *origination information*, as it is information concerned with on which strands information originates. The value of the nonce originates on one strand so we say it was uniquely originating. The private decryption keys are not found on any



(a) Shape consisting of Responder 1 transmitting the nonce



(b) Shape consisting of Responder 2 transmitting the nonce

Figure 4: The shapes resulting from a skeleton observing $a \downarrow 3$ when the nonce is uniquely originating and private keys of B and C are uncompromised

of the messages in the strands and are therefore non-originating. To better understand how origination information affects shape discovery, consider the case where no unique or non-origination information is attached to the query. CPSA generates a single shape realizing the third node of the initiator without involving other strands as shown in Figure 5. The realized shape looks similar to the strand definition, showing only that the messages were sent and received. How can this be? Recall that shapes involve only regular strands, or strands that are part of the protocol definition itself. Regular strands do not include the actions of adversaries. If the nonce was not uniquely originating or if the private decryption keys were compromised, the observed behavior could be achieved with adversary behavior alone, such as an adversary who knows how to decrypt one or both of the first messages or possess the nonce in advance. No additional regular strands are involved in the shape to adequately describe the skeleton observation. The penetrator strands that implicitly represent adversarial behavior may have received one of the two messages sent and decrypted it or simply had prior knowledge of the nonce since its value was not assumed to be uniquely originating on the initiator strand. Both of these possible scenarios suffice to explain the skeleton without the involvement of regular strands.

Message contents may include information that is either *uniquely-originating* or *non-originating*. Uniquely-originating data originates only in one strand. For instance, a nonce

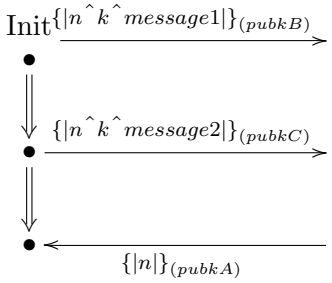


Figure 5: The single shape resulting from a skeleton observing $a \downarrow 3$ when the nonce is not uniquely originating and private keys for B and C are compromised

is presumed to be a fresh value generated by a particular participant. Such a value is unguessable (again, in a perfect cryptosystem) and is not repeated in future executions of the protocol. Therefore a participant transmitting a certain piece of uniquely-originating data on a strand other than the one that it originated on must have received this data on some previous node. Similarly, data that is non-originating appears on no strands. A private decryption key, for instance, is not known to any other participant and therefore messages signed with that key could not have been forged and messages encrypted with the private key's corresponding public key could not be decrypted by an adversary.

2.5 Penetrator Strands

Penetrator strands represent the behavior of adversaries. Like regular strands, penetrator strands are composed of a trace mapping for messages sent and received. Penetrator strands do not make up part of the protocol definition as they are assumed to be unbound in both their number and possible configurations they can take while abiding by the origination constraints placed on the protocol definition.

Penetrator strands, then, are limited in their capabilities: they can only decrypt messages for which they possess the decryption keys for and sign messages for which they possess the necessary private keys. These penetrators can still perform such operations as concatenation of known information, encrypting message contents and so forth.

Although the output of CPSA does not include penetrator strands, their behavior is nonetheless reflected by the realization of regular strand reception nodes without the expected matching transmission nodes of another regular strand. We saw in Figure 5 that this may be the result of where information in the messages originate. As a slightly different example, consider one possible shape generated from CPSA realizing the final

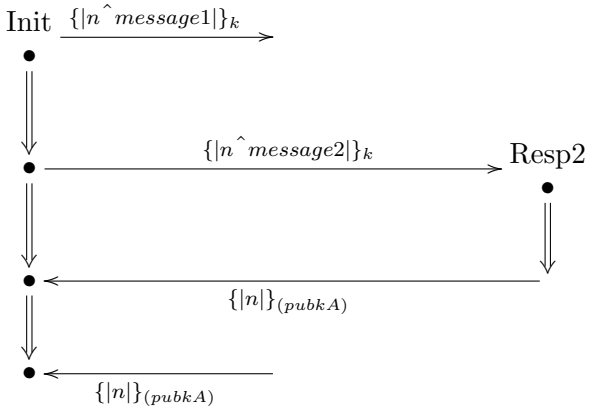


Figure 6: One possible shape realizing $a \downarrow 4$. The realization of this node without a matching regular strand's transmission hints to adversarial activity

node of the initiator strand when the nonce is uniquely originating and the private keys are non-originating (uncompromised) as shown in Figure 6. The final reception node of the initiator is realized without involving any further regular strands. The explanation of this output comes from considering the ability of an adversary to repeat messages it observes being transmitted even though it cannot decrypt the message or understand its importance in any way. Note that the messages received on the third and fourth node of the initiator strand are identical. Even though such an adversary was not able to decrypt the message and view its content, it could simply relay the message allowing the realization of this node without further interaction from either responder.

2.6 Rely-Guarantee Method

A protocol definition is concerned only with the form of messages sent and received by principals; it says nothing of the implications of sending and receiving messages, despite the fact that such messages have concrete interpretation within the protocol. In a cryptographic money-order protocol, for instance, a bank may commit to transferring funds from the client to any merchant should the client authorize the transfer and some merchant request the transfer. This commitment by the bank would take place the moment the bank issues the unendorsed money order to the client. Similarly, when this client adds a cryptographic signature to the money order, thereby endorsing it, the client has committed to authorizing the transfer.

To reflect the real-world implications tied to message transmissions and receptions, CPPL and CPSA adopt a rely-guarantee method of annotating the various nodes of strands

that make up a protocol definition. Each message sent can be annotated with a *guarantee* and each message received with a *rely*. These formulas may take on any form as their meaning is significant only within the protocol. Whenever a principal receives a message whose corresponding node is annotated with a rely, that principal may use the information provided in that rely to determine possible branches in execution, such as whether or not to continue executing the protocol. Such rely formulas bridge the gap between the protocol definition itself and the trust management systems of the principal executing the protocol. Guarantees are written as some expression ϕ and relies are written as `P says` ϕ to express that a principal `P` uttered the guarantee ϕ. It is up to the trust system of the receiving principal to determine whether or not to conclude ϕ based on the rely `P says` ϕ.

A protocol is correctly annotated if on each node to which a rely formula was added, there exists a corresponding guarantee. Thus a correctly annotated protocol proscribes exactly what information a principal can use upon message reception to determine its own branching behavior.

Penetrator strands have none of their nodes annotated with rely or guarantee formulas; this reflects that adversaries do not make commitments that have any meaning in the protocol execution [3]. This illustrates that a correctly annotated and sound protocol would not allow a regular strand to rely on a particular formula `P says` ϕ purely on the fact that it received a particular message. If an adversary was allowed to construct that message itself (say, using a previously compromised key) the receiving principal would not be able to be certain that ϕ was uttered by principal `P`. A protocol can only be sound if each node containing a rely such as `P says` ϕ is preceded by the principal `P` uttering the guarantee ϕ on every possible execution of the protocol.

3 Syntax of CPSA

CPSA uses an S-expression syntax for both defining protocols and skeletons (queries) to be expanded into shapes. S-expressions take the form of nested sets of parentheses with values separated by whitespace.

3.1 Defining protocols

Protocols are defined using the `defprotocol` keyword. CPSA currently supports two grammars. The grammar used throughout this paper is the *basic* grammar. Besides a name and the `basic` keyword used to indicate the basic grammar is in use, a protocol consists of definitions for each of the regular strands that may participate in a particular run of a protocol.

The structure of an incomplete protocol definition in the basic grammar is shown below. In an actual protocol definition, role definitions would follow in place of the square-bracketed text and ellipsis.

Listing 1: An incomplete protocol definition

```
(defprotocol protocol_name basic
  [role defintion]
  [...]
)
```

Each role is defined using the `defrole` keyword. A role is made up of a set of variables to be used in that role, a trace (list) of message transmissions and receptions, and optionally a set of annotations. Variables belong to sets such as `name`, `text`, `data`, and `skey` (shared key). Traces defined with the `trace` keyword are made up of an implicitly ordered set of `send` and `recv` expressions for message transmissions and receptions respectively. The arguments following each `send` and `recv` represent the form of the contents of the message. The message contents may consists of any combination of variables or literals combined with the concatenation and cryptographic operations such as encryption.

In the incomplete protocol definition below, a role with name *Alice* is defined which consists of a set of variables and a trace. The variable line indicates two names a and b, a piece of text labeled *some_data*, and a shared key labeled *a_key*. In the first message, Alice sends the *some_data* ingredient along with the key *a_key* encrypted with the public

key of b. Next Alice receives the same ingredient of data back encrypted with the shared key previously transmitted. In an actual protocol definition, additional roles would be specified, such as the role represented by the name b in this strand.

Listing 2: Protocol definition with a single role

```
(defprotocol protocol_name basic
  (defrole Alice
    (vars (a b name) (some_data text) (a_key skey))
    (trace
      (send (enc some_data a_key (pubk b)))
      (recv (enc some_data a_key))
    )
   [...]
)
```

Lines 5 and 6 of Listing 2 show the operation of encryption, as denoted with the `enc` keyword. The last argument to this function is the key to be used. Symmetric and asymmetric key encryption are performed using the same `enc` keyword. In the example, the first use of `enc` with key `(pubk` b`)` represents asymmetric key encryption. A participant possessing `(privk` b`)` can decrypt this message. The second use of encryption with the key a_key is symmetric. It follows that any participant possessing a_key can decrypt the second message.

Lastly, optional relies and guarantees may follow the `annotation` keyword and consist of numbers indicating the 0-based node of the strand the annotation is associated with followed by the annotation itself. Formulas attached to transmission nodes are guarantees and formulas attached to reception nodes are relies. Relies are written in the form of `P says` ϕ to reflect that guarantees such as ϕ are uttered by principals. Recall that this notation says nothing of whether or not the receiving principal trusts what was said by the principal making the utterance.

In Listing 3, the two nodes of the Alice stand are annotated.

Listing 3: Protocol definition with annotations

```
(defprotocol protocol_name basic
  (defrole Alice
    (vars (a b name) (some_data text) (a_key skey))
    (trace
      (send (enc some_data a_key (pubk b)))
      (recv (enc some_data a_key))
    )
    (annotations a
      (0 (guarantee1))
      (1 (says b (guarantee2)))
    )
)
```

On the first node (a transmit node) *Alice* is making the guarantee *guarantee*1, signifying that the guaranteed expression is true when the first message is transmitted. The second node (a receive node) has a rely that indicates by receiving that message, principal *b* has uttered the guarantee *guarantee*2.

3.2 Defining Skeletons

When a protocol designer wishes to query CPSA for behavior compatible with a particular message, the designer must first create a skeleton that represents that some message transmission or reception has taken place and pass this along with the protocol definition to CPSA. CPSA will expand the skeleton in all of the essentially different ways possible under the constraint information found in the skeleton. Recall that this process will generate the different shapes of the protocol execution that explain the behavior of the skeleton. The shapes allow the protocol designer to observe the possible ways in which a protocol can execute.

A skeleton is defined with the `defskeleton` keyword. Within the `defskeleton` are declarations of the name of the protocol being instantiated, variables to be used, strand definitions, and origination constraint information. Strands are defined with the `defstrand` keyword and consist of a strand name, 1-based node number, and variable bindings. The bindings map from the variables of the skeleton to the variables of the `defrole` in the corresponding protocol definition. The `defstrand` represents which node of the strand needs to be justified by CPSA. Optional origination information may follow using the `non-orig` and `uniq-orig` keywords.

Listing 4: A skeleton for the epmo protocol

```
(defskeleton epmo (vars (b c m name))
  (defstrand bank 1 (b b) (c c) (m m))
  (non-orig (privk b) (privk c)))
```

In Listing 4, a skeleton is shown for a protocol named *epmo* that will be explored in Section 5 of this paper. The skeleton defined in this listing shows that the first node of a bank strand has taken place and specifies that the private keys of b and c do not originate on any strand of the protocol. In this paper, since skeletons used as queries will always contain only one `defstrand`, the variable bindings will consist only of pairs of each variable found in the skeleton.

4 Goals

This project attempted to solve two problems regarding the usefulness of CPSA. First, it is possible for a protocol designer to create an unsound rely-guarantee formula assignment by performing this process by hand. CPSA would be more useful if an algorithm could correctly assign rely formulas from an assignment of guarantee formulas. Second, the branching information of non-originating and uniquely-originating data should not reside as part of the protocol definition or a specific query to CPSA. Instead this branching information should somehow be meshed with the run-time language of the relies and guarantees. This will allow a more congruent interaction between the two pieces of information responsible for branching behavior in the participants.

4.1 Generation of rely formulas

Leaving the protocol designer to annotate both the rely and guarantee formulas for all the principals allows her to make these assignments in an unsound way. For instance, the protocol designer may annotate a particular receiving node `n` with formula $\texttt{P}_\texttt{i}$ `says` $\phi_\texttt{i}$ even though there exists a shape realizing a skeleton observing `n` where $\texttt{P}_\texttt{i}$ had not uttered $\phi_\texttt{i}$. If this mistake was allowed to go unchecked, the principal receiving the message may go on to make branching decisions based on unsound data, that is, data that does not reflect the shared state of all principals.

It would be better if the protocol designer merely had to annotate what is guaranteed by each message transmission. An algorithm would then use CPSA to determine the guarantees that must have preceded each message reception. Such a change would ensure that protocols are always annotated in a sound fashion. It would also provide a sort of check for the protocol designer to see that the anticipated information is placed as relies on the relevant nodes. If the protocol designer expected a participant to be able to make a decision at a particular node and the algorithm's output shows the necessary information to make that decision is not present as a rely, the protocol designer can investigate how to modify the protocol to ensure that rely at the time of message reception.

4.2 Restructuring origination information

As it stands, information regarding where message contents and keys originate is provided by a set of keywords attached to the protocol definition or a specific query for

CPSA. In practice, this syntax is used to denote, for instance, that a certain principal's private decryption key is not compromised (non-originating with respect to all principals participating) or that a certain nonce appearing in a message was freshly generated and not guessable (uniquely-originating from one principal and known at first only by that principal).

Data regarding the unique-origin or non-origin of items such as message nonces or keys is useful for determining the branching behavior of particular principals at runtime. Such information should therefore be placed with other runtime branching information such as the rely-guarantee formulas. This project will seek to expand the language of the rely-guarantee formulas to accurately represent this origination data in a manner that is semantically congruent with formulas used to describe protocol shapes.

5 Rely Generation

Since a sound assignment of relies and guarantees means that relies are always matched with corresponding guarantees when a protocol is executed, it makes sense for a protocol designer to annotate only the guarantees attached to each message transmission. An algorithm can then determine the guarantees that must have been uttered by the time a particular message is received.

A principal may guarantee that some expression ϕ is true when sending a message. Relies on reception nodes that must have been preceded by such a guarantee should reflect this necessary ordering. This is not to say that the principal receiving the message should be able to directly conclude ϕ. The guarantee must state which principal uttered the expression. For this reason relies make use of the `says` verb, to indicate not that a formula is true, but that some other participant believes (or wants the receiving principal to believe) that the formula is true.

This notion of relies signifying only the utterance of guarantees codifies the fact that not all information from all participants may be trusted. Trust information maintained by each principal separately is combined with relies to derive knowledge necessary to determine what message, if any, should be sent next. Such trust information is not described here as it could take many forms and may persist over protocol executions and change over time. Trust information may include any local data store the principal has access to. If one principal was in the business of selling access to a paid service, such information might include subscription lists for paid services, or perhaps a record of which subscribers have made late payments in the past and therefore are more likely to make late payments in the future.

Eventually, a particular principal is likely to make use of its local data store to determine whether something uttered by another principal is true. For example, if a particular participant has as part of its trust management system the expression `For all` ϕ `if P says` ϕ`, then` ϕ and has received a message on a node with rely `P says moneyTransferred`, the participant in question can use these two pieces of information together to conclude that payment for the product in question has been transferred and that, perhaps, it is now safe to ship the product that was being sold.

5.1 Algorithm Design

An algorithm to automatically assign relies from a set of guarantees must ensure that each reception node is annotated with no further assumptions about which message transmissions or receptions have also occurred. Recalling that a skeleton can be used as a query to CPSA to probe for the essentially different ways participant behavior can result in a particular node taking place, a query can be used to derive the different possible orderings of nodes necessary to explain one particular node. This observation hints to a solution utilizing a skeleton to query each node to be annotated individually. As each shape produced by the query reflects the essentially different executions that could explain the skeleton, we know that during execution exactly one of the shapes has occurred when a message is received. Once the nodes preceding the node to be annotated have been determined, the annotations associated with these nodes can be extracted from the protocol definition.

After extracting the relies associated with nodes known to precede a node being annotated, one possible approach for combining this information is to take the intersection of the guarantees that must have been uttered for each shape realized. Although this intersection would necessarily represent guarantees that are always uttered before message reception, this intersection could likely be the empty set. Although an assignment of the empty set to the rely formula of any particular node would be true, it would not be very useful in allowing the receiving principal to conclude anything more interesting than what it could conclude before the message was received.

As a compromise, the algorithm proposed here makes clear the notion that it is possible for different sets of participant behavior to precede a particular node. In the event only a single shape was realized for a skeleton querying a node, the result is an unambiguous statement of what nodes (and relies) preceded the node. If two or more shapes are produced from the query, the output of the algorithm reflects the fact that there are multiple sets of relies that may have preceded the node. For the nodes that can be explained by multiple shapes, the output preserves the ability of the receiving participant to use the information gleaned from the rely in any manner it sees fit, whether this means taking the intersection of the possible sets of compatible behavior or not.

5.2 Algorithm Implementation

The algorithm presented here takes as input a protocol definition and a set of shapes produced by CPSA in the S-expression input/output language of CPSA. The shapes are

the result of CPSA expanding query skeletons written by the protocol designer. By creating skeletons observing each reception node in the protocol, the algorithm will extract the guarantees that must have preceded each reception node and generate the appropriate relies based on the shapes produced by CPSA. The algorithm produces as output the S-expression protocol definition augmented with rely formulas on each receiving node queried for in the skeletons used as input to CPSA.

The algorithm enumerates the reception nodes and determines the annotations that can be assigned to them. It does so by constructing a disjunct of conjuncts, each disjunct representing one shape derived from the skeleton for a particular message reception and each conjunct representing a guarantee observed in that shape. The guarantees are determined by inspecting the list of "precedes" (pairs representing ordering of messages) generated by CPSA. These pairs are searched to reconstruct all possible message transmissions that must have occurred before the message reception being annotated.

5.2.1 Determining message order

For each shape processed, the algorithm must first determine all message transmissions preceding the queried strand. This can be accomplished by utilizing two pieces of information together.

First, as a strand is a trace sequence of message transmissions and receptions, for any strand s with nodes n and $n-1$, if node n has been observed then node $n-1$ must also have been observed. This information may be extracted from the individual strand trace definitions. As a protocol is made up of a set of strands, this ordering holds for all shapes of that protocol.

Second, as a shape is a partial ordering of nodes, CPSA produces a *precedes* line as part of the skeleton-to-shape expansion. The precedes line provides the information concluded by CPSA regarding which nodes precede which other nodes under the origination constraints placed on the query. The precedes line takes the form of a set of pairs of nodes. Each pair $(n1, n2)$ representing the fact that node $n2$ is preceded by node $n1$.

To combine these two pieces of information the precedes line is searched to determine nodes that must have occurred before the node being annotated. At start, the algorithm knows only of the node n being annotated and can check the precedes line to see if it contains a pair ($precedingNode$, n). If so the node $precedingNode$ is also checked for in the precedes line to determine further nodes preceding n. Additionally, each time a node

is checked for in the precedes line, the node occuring before that node on the same strand can also be checked for.

In the implementation, a list is maintained containing the working set of all nodes that precede the goal node. Assume that the query node being annotated is *qnode* and the precedes list item $(n1, n2)$ represents that node $n1$ precedes node $n2$ in the shape and that all nodes are a $(strand, nodeNumber, annotation)$ tuple. The algorithm is shown in Algorithm 1.

Algorithm 1 annotationsBefore(a node *qnode*, a shape *shape*) → list of annotations

```
Let answer be a list of annotations
Let inProgress be a stack of nodes
inProgress.push(qnode)
while inProgress is not empty do
  n = inProgress.pop()
  if n is a transmit node then
    answer.add(n.annotation)
  end if
  for all (n1, n) in shape.precedes do
    inProgress.push(n1)
  end for
  if n.nodeNumber > 0 then
    inProgress.push(nodePreceding(n))
  end if
end while
return answer
```

By the end, the list *answer* contains all the relevant annotations preceding the original node *qnode*. The implementation as described above assumes no changes in the veracity of guarantees over time. Once a principal makes a guarantee, that guarantee is assumed to hold until the end of the protocol's execution.

5.2.2 Constructing relies

Utilizing Algorithm 1 for determining all the guarantees that precedes a node in a particular shape, the generation of sound relies is straightforward from input containing the result of CPSA processing skeletons querying each receiving node in the protocol definition. The resulting rely assigned to each transmission node is (at most) a disjunction of conjunctions. Each disjunct represents one of the shapes realized by CPSA and each conjunct is a rely in the form `P says` ϕ where ϕ was asserted by some guarantee on a node preceding the reception node. The algorithm is shown as Algorithm 2.

Algorithm 2 annotateNodes(list of nodes *nodes*, list of shapes *shapes*) → list of nodes

let *answer* be a list of annotated nodes
for all n in *nodes* **do**

$$n.annotation = \left(\bigvee_{s \text{ realizing } n \in shapes} \left(\bigwedge_{g \in \text{ annotationsBefore}(n,s)} g \right) \right)$$

answer.add(n)
end for
return *answer*

For protocols containing one or fewer shapes, or shapes in which one or fewer relies were observed to precede a transmission node, the formula may be further simplified utilizing the logical rules for disjunction and conjunction.

5.3 Algorithm Analysis

5.3.1 Termination

Proving that Algorithm 2 terminates is simple since it calls Algorithm 1 once for every node passed as input. Therefore we only need to prove that Algorithm 1 always terminates. The proof of this follows.

Sketch of the proof: In any shape used as input to the algorithm, there exists a finite number of strands $S_1 ... S_n$. Each strand S_i has a finite number of nodes $S_{iN_1} ... S_{iN_m}$ The total number of nodes is the product of two finite terms and is therefore also finite.

The algorithm terminates when the *inProgress* stack becomes empty. At start this stack contains exactly one node. In each iteration of the main loop, the algorithm first pops a node off of the stack and then may optionally add one or more nodes to the stack. The nodes to be added are from one or both of two sets. A node added may be from the precedes list or a node in the strand preceding the node.

A node added to the stack from the first set states that a node on a strand a is preceded by a node on strand b. Since the precedes list represents a partial ordering of nodes, in each iteration of the loop processing the new node, a node will never be added that was observed in execution after the current node.

A node added to the stack from the second set is a node on strand a that precedes

the node on strand a. Similarly, in each iteration of the loop processing this new node, a node will never be added that is found after the current node on the strand.

Since there are a finite number of nodes and the result of processing a node in each iteration against each of the sets adds only nodes occurring before the node being processed, the possible number of nodes to be added to the stack decreases in each iteration. The total number of nodes to be added to the stack is therefore finite, as it is the sum of decreasing integers. Since the total number of nodes to be added to the stack over all iterations is finite and one node is popped from the stack on each iteration, the stack will eventually be empty and the algorithm will terminate.

5.3.2 Soundness

An assignment of relies and guarantees is sound if every guarantee is true in all executions of the protocol.

Sketch of the proof: Since the guarantees assigned are in the form of a disjunction of conjunctions, with each disjunction resulting from one shape of the protocol's execution, and since exactly one of the shapes must be observed in any protocol execution, the algorithm is sound if each conjunction is sound for at least one shape.

Each guarantee in each conjunction must have been uttered in the shape used to derive that conjunction before the rely in order for the algorithm to be sound. For each node queried as part of a skeleton expansion, the precedes list is checked for nodes preceding the goal node. Each node read from the precedes list has been uttered, and the burden of proof falls on CPSA. Since a trace ordering dictates a transitive relation between all nodes observed, any node resulting from the precedes list can also be checked against the precedes list. Since strands are linear, if a node S_{iN} has been observed then every node $S_{iN_1}...S_{iN_{n-1}}$ has also been observed.

5.4 Algorithm Architecture

The algorithm is implemented as a set of three Python scripts. One script is responsible for parsing the S-expression output of CPSA into nested data structures (Python classes). A second script developed by Elliott Franco Drabek and placed in the public domain is responsible for the low-level parsing of S-expressions into Python lists and atoms, somewhat analogous to the Scheme read function. The last script is responsible for actually

performing the steps of the algorithm. See Section A for code listing.

5.5 Example Execution

5.5.1 Basic Example

Consider a protocol in which an initiator sends two distinct messages to two distinct responders, similar in spirit to Figure 3. Each message contains the same uniquely originating nonce. These responders then respond with that nonce. We would expect that although the initiator knows that each response came from one of the responders, the initiator would not be able to match responses to responders. If we encode the three strands in the CPSA input language and annotate the responders message transmissions we might do so as shown in the following snippet:

Listing 5: An example protocol with guarantees

```
(defprotocol example1 basic
  (defrole init
    (vars (a b c name) (k n text))
    (trace
      (send (enc n a "text1" (pubk b)))
      (send (enc n a "text2" (pubk c)))
      (recv (enc k n (pubk a)))
      (recv (enc k n (pubk a)))
    )
  )
  (defrole resp1
    (vars (a b c name) (k n text))
    (trace
      (recv (enc n a "text1" (pubk b)))
      (send (enc k n (pubk a)))
    )
    (annotations b
      (1
        (responder_1_repeats_the_secret)))
  )
  (defrole resp2
    (vars (a b c name) (k n text))
    (trace
      (recv (enc n a "text2" (pubk c)))
      (send (enc k n (pubk a)))
    )
    (annotations c
      (1
        (responder_2_repeats_the_secret)))
  )
)
```

Listing 6: Continued from Listing 5: example protocol query skeletons

```
;;Query node 3 of strand a
(defskeleton example1
  (vars (a b c name) (k n text))
  (defstrand init 3 (a a) (b b) (c c) (n n) (k k) )
  (non-orig (privk a) (privk b) (privk c))
  (uniq-orig n)
)

;;query node 4 of strand a
(defskeleton example1
  (vars (a b c name) (k n text))
  (defstrand init 4 (a a) (b b) (c c) (n n) (k k) )
  (non-orig (privk a) (privk b) (privk c))
  (uniq-orig n)
)
```

The value of the annotation for each responder is written as to be descriptive of the action being performed on that message transmission. For instance, the guarantee "responder_1_repeats_the_secret" indicates that the transmission of that message means that the first responder has generated a message containing the nonce. The second responder is similarly annotated. In this way, when we analyze the automatically-generated rely formulas for nodes on the initiator strand receiving the nonce back, we can read off from the annotation which responder was responsible for repeating the secret nonce.

CPSA will use the two skeletons present in the input as starting points to expand into realized shapes. These skeletons represent the observation of a particular node on the strand (counting from one) and include the origination constraints to place on the query. In both queries, the nonce is fixed as a uniquely-originating value and each private key is fixed as a non-originating value (representing an uncompromised key). CPSA's output from processing these two queries can be passed to the rely-generation algorithm, which in turn emits a new protocol definition with relies added to the two message receptions on the initiators strand that were originally queried.

Listing 7: Example protocol after rely assignment

```
(defprotocol example1 basic
  (defrole init
    (vars (a b c name) (k n text))
    (trace (send (enc n a "text1" (pubk b)))
      (send (enc n a "text2" (pubk c))) (recv (enc k n (pubk a)))
      (recv (enc k n (pubk a))))
    (annotations
      (2
        (or (says b (responder_1_repeats_the_secret))
          (says c (responder_2_repeats_the_secret))))
      (3
        (or (says b (responder_1_repeats_the_secret))
          (says c (responder_2_repeats_the_secret))))))
  (defrole resp1
    (vars (a b name) (k n text))
    (trace (recv (enc n a "text1" (pubk b))) (send (enc k n (pubk a))))
    (annotations b (1 (responder_1_repeats_the_secret))))
  (defrole resp2
    (vars (a c name) (k n text))
    (trace (recv (enc n a "text2" (pubk c))) (send (enc k n (pubk a))))
    (annotations c (1 (responder_2_repeats_the_secret)))))
```

The output properly reflects the fact that each message reception could have been due to the reception of the nonce from either of the responders. For this reason, each of the last two nodes on the initiator strand now contain the expression `(or (says b (responder_1_repeats_the_secret)) (says c (responder_2_repeats_the_secret)))`. It is important to note that the annotations reflect *what* did happen, but not necessarily *how*. Recalling Figure 6, we know that the shape generated for the observation of the last node of the initiator strand actually includes adversarial behavior. Adversarial behavior in this case does not change the output as one of the responders still must have sent the corresponding message. The relies reflect the greatest amount of information that can be reliably known from other principals but says nothing of, for instance, what strand (legitimate or penetrator) was responsible for the message that was received.

This example also illustrates the degenerative form of the automatically generated relies for when only one guarantee is made for each shape. Generally, automatically generated relies take the form of a disjunction of conjunctions. In this example, since each shape produced only a single annotation, we are left with simply a disjunction. Each disjunct represents a guarantee that was uttered by some principal during the realization of that shape.

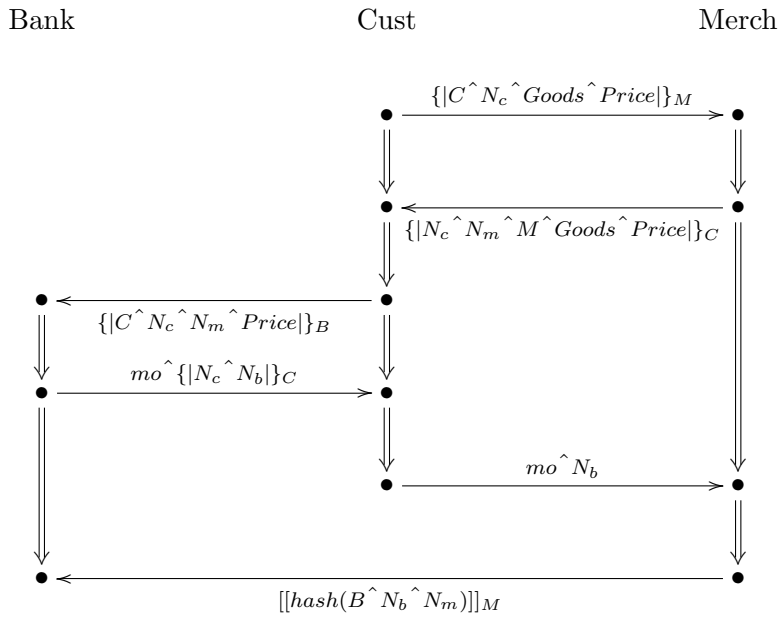


Figure 7: The EPMO protocol (money order $mo = [[hash(C \hat{} N_c \hat{} N_b \hat{} N_m)]]_B$)

5.5.2 EPMO example

In a more complicated example, the rely-generation algorithm may be used along with CPSA to annotate multiple principals in a protocol definition. The protocol definition in Listing 8 and queries in Listing 9 will serve as input to generate relies for a protocol named "Electronic Purchases using a Money Order". An execution of this protocol is shown in Figure 7.

This protocol allows a customer to agree on a price with a merchant for some set of goods. The customer may then have a bank reserve funds from an account as payment to the merchant. The bank issues a money order that can be endorsed by the customer and cashed by the merchant in exchange for shipping the goods.

In such an exchange, each participant needs to be sure exactly what they are committing to by sending a particular message and conversely what they learn about other participants upon receiving a message of particular ingredients. The rely-guarantee method allows these trust issues to be encoded in a manner that could be combined with each participant's trust policies. Not shown in an annotated protocol would be information such as the merchant knowing the bank is reliable, the customer utilizing a database of "good merchants" to decide to do business with this merchant and other pieces of information that may be part of the trust management policy used by each principal.

The annotations express statements such as the merchant's promise to ship goods should any bank transfer the funds to the merchant's account. Similarly, by issuing a money order the bank promises that should the customer authorize the transfer of funds (by endorsing the money order) and should the merchant request the transfer (by cashing the money order), the bank will in fact transfer the reserved funds from customer to merchant.

Such promises can be seen in the annotations added to the send nodes of the strands making up EPMO. Similarly, relies (whether hand written or automatically generated) should express the unknown veracity of each guarantee with the `says` keyword. For this example, relies have been borrowed from the paper [4] from which this protocol was presented in.

Listing 8: EPMO with guarantees

```
;;; Electronic Purchase with Money Order protocol annotated with trust
;;; management formulas.

(defprotocol epmo basic
  (defrole bank
    (vars (b c m name) (hash akey) (nc nm nb data) (price text))
    (trace
     (recv (enc c nc nm price (pubk b)))
     (send (cat (enc (enc c nc nb nm price hash) (privk b))
                (enc nc nb (pubk c))))
     (recv (enc (enc b nb nm hash) (privk m))))
    (non-orig (invk hash))
    (uniq-orig nb)
    (annotations b
      (1
        (forall ((pm name))
          (implies
            (and (says c (transfer b price pm nm))
              (says pm (transfer b price pm nm)))
            (transfer b price pm nm))))
      ))
  (defrole customer
    (vars (b c m name) (hash akey) (nb nc nm data) (goods price text))
    (trace
     (send (enc c nc goods price (pubk m)))
     (recv (enc nc nm m goods price (pubk c)))
     (send (enc c nc nm price (pubk b)))
     (recv (cat (enc (enc c nc nb nm price hash) (privk b))
                (enc nc nb (pubk c))))
     (send (cat (enc (enc c nc nb nm price hash) (privk b)) nb)))
    (non-orig (invk hash))
    (uniq-orig nc)
    (annotations c
      (4 (transfer b price m nm))))
  (defrole merchant
    (vars (b c m name) (hash akey) (nb nc nm data) (goods price text))
    (trace
     (recv (enc c nc goods price (pubk m)))
     (send (enc nc nm m goods price (pubk c)))
     (recv (cat (enc (enc c nc nb nm price hash) (privk b)) nb))
     (send (enc (enc b nb nm hash) (privk m))))
    (non-orig (invk hash))
    (uniq-orig nm)
    (annotations m
      (1
        (forall ((pb name))
          (implies (transfer pb price m nm) (ship m goods c))))
      (3 (and (transfer b price m nm) (ship m goods c))))))
```

Listing 9: Continued from Listing 8: EPMO query skeletons

```
;;Query node 0 of the bank strand
(defskeleton epmo (vars (b c m name) (goods name))
  (defstrand bank 1 (b b) (c c) (m m))
  (non-orig (privk b) (privk c)))

;;Query node 2 of the bank strand
(defskeleton epmo (vars (b c m name) (goods name))
  (defstrand bank 3 (b b) (c c) (m m))
  (non-orig (privk b) (privk c) (privk m)))

;;Query node 1 of the customer strand
(defskeleton epmo (vars (b c m name))
  (defstrand customer 2 (b b) (c c) (m m))
  (non-orig (privk c) (privk m)))

;;Query node 3 of the customer strand
(defskeleton epmo (vars (b c m name))
  (defstrand customer 4 (b b) (c c) (m m))
  (non-orig (privk b) (privk c) (privk m)))

;:Query node 0 of the merchant strand
(defskeleton epmo (vars (b c m name))
  (defstrand merchant 1 (b b) (c c) (m m))
  (non-orig (privk c) (privk m) ))

;;Query node 2 of the merchant strand
(defskeleton epmo (vars (b c m name))
  (defstrand merchant 3 (b b) (c c) (m m))
  (non-orig (privk b) (privk c) (privk m)))
```

The automatic rely-guarantee generation algorithm produces the following result:

Listing 10: EPMO after rely assignment

```
(defprotocol epmo basic
  (defrole bank
    (b c m name)
    (hash akey)
    (nc nm nb data)
    (price text)
    (trace (recv (enc c nc nm price (pubk b)))
      (send
        (cat (enc (enc c nc nb nm price hash) (privk b))
          (enc nc nb (pubk c))))
      (recv (enc (enc b nb nm hash) (privk m))))
    (annotations b (0 true)
      (1
        (forall ((pm name))
          (implies
```

```
          (and (says c (transfer b price pm nm))
            (says pm (transfer b price pm nm)))
          (transfer b price pm nm))))
    (2
      (and
        (says m
          (forall ((pb name))
            (implies (transfer pb price m nm) (ship m goods c))))
        (says m (and (transfer b price m nm) (ship m goods c)))
        (says c (transfer b price m nm))))))
(defrole customer
  (b c m name)
  (hash akey)
  (nb nc nm data)
  (goods price text)
  (trace (send (enc c nc goods price (pubk m)))
    (recv (enc nc nm m goods price (pubk c)))
    (send (enc c nc nm price (pubk b)))
    (recv
      (cat (enc (enc c nc nb nm price hash) (privk b))
        (enc nc nb (pubk c))))
    (send (cat (enc (enc c nc nb nm price hash) (privk b)) nb)))
  (annotations c
    (1
      (says m
        (forall ((pb name))
          (implies (transfer pb price m nm) (ship m goods c)))))
    (3
      (and
        (says m
          (forall ((pb name))
            (implies (transfer pb price m nm) (ship m goods c))))
        (says b
          (forall ((pm name))
            (implies
              (and (says c (transfer b price pm nm))
                (says pm (transfer b price pm nm)))
              (transfer b price pm nm))))))
    (4 (transfer b price m nm))))
(defrole merchant
  (b c m name)
  (hash akey)
  (nb nc nm data)
  (goods price text)
  (trace (recv (enc c nc goods price (pubk m)))
    (send (enc nc nm m goods price (pubk c)))
    (recv (cat (enc (enc c nc nb nm price hash) (privk b)) nb))
    (send (enc (enc b nb nm hash) (privk m))))
  (annotations m (0 true)
    (1
      (forall ((pb name))
        (implies (transfer pb price m nm) (ship m goods c))))
    (2
      (and
```

```
    (says b
      (forall ((pm name))
        (implies
          (and (says c (transfer b price pm nm))
            (says pm (transfer b price pm nm)))
          (transfer b price pm nm))))
    (says c (transfer b price m nm))))
(3 (and (transfer b price m nm) (ship m goods c))))))
```

In this more complicated example, it is worth looking at each automatically-generated annotation and comparing this to how the authors of the protocol annotated it in their paper in order to be sure the two agree logically.

On the final node of the bank strand, a human might annotate to indicate that on message reception, the bank learns that both the customer and the merchant have requested the bank to transfer the amount of money agreed on. Listing 11 captures this as a formula in CPSA syntax.

Listing 11: Node 3 of bank strand in EPMO

```
(and (says c (transfer b price m nm))
  (says m (transfer b price m nm)))
```

The generated rely reflects that this same fact must have occurred but also introduces other guarantees that must have happened. As the algorithm has no knowledge of how the relies will be used, it presents more information than what the bank would be concerned with in this implementation. Here, we see that the bank knows information about the merchant, such as that the merchant will ship goods if any bank transfers the funds. Although not needed, we know that such statements are true since they must have happened in the one shape that could have possibly resulted in this message reception by the bank. Unlike the previous example, because only a single shape was produced, no disjunct exists at the top level of any of the annotations. Instead each annotation is made up of a conjunction of all preceding guarantees for the one single shape observed.

For the customer, on the first receiving node, a human annotator would likely indicate that by receiving a confirmation on the price of goods, the seller agrees to ship the goods after payment has been made by any bank as promised by the merchant in its guarantee.

Listing 12: Node 2 of customer strand in EPMO

```
(says m
  (forall ((pb name))
    (implies (transfer pb price m nm) (ship m goods c))))
```

This is the exact result produced by the algorithm since in this case it was the only guarantee observed in the single shape produced by CPSA.

When the customer receives the "money order" on the next reception, we expect it to learn that the bank has promised to pay any merchant should that merchant request the transfer and should the customer authorize it as in Listing 13.

Listing 13: Node 4 of customer strand in EPMO

```
(says b
  (forall ((pm name))
    (implies
      (and (says c (transfer b price pm nm))
        (says pm (transfer b price pm nm)))
      (transfer b price pm nm))))
```

Here too the algorithm aligns with expectations, adding that the merchant's promise to ship the goods has been uttered, as already determined on its previous message reception.

On the last message received by the merchant (the endorsed money order), a likely annotation would indicate that the merchant knows the bank will transfer money to any merchant providing that the customer has authorized the transfer and the merchant has requested it. Such an annotation would also express that the merchant knows the customer has requested the transfer, thus the bank has promised the transfer to take place providing the money order is cashed. The below annotation would be all that is needed to express these facts.

Listing 14: Node 3 of merchant strand in EPMO

```
(and
  (says b
    (forall ((pm name))
      (implies
        (and (says c (transfer b price pm nm))
          (says pm (transfer b price pm nm)))
        (transfer b price pm nm))))
  (says c (transfer b price m nm)))
```

As we would expect the annotations emitted by the algorithm align with these expectations.

5.6 Discussion

Although the algorithm produces a sound assignment of rely formulas, it is clear from the above example that often times these formulas are more detailed than necessary. Fortunately, such additional information does not subtract from the programmatic usefulness of the relies with respect to their implementation on distributed machines.

The algorithm is currently limited to processing protocols written in the *basic* grammar and must be used in conjunction with CPSA in order to process skeletons into shapes. As the algorithm is implemented completely separately from CPSA, there is a small amount of computational overhead in reparsing the protocol and reparsing the shapes.

What is not clear from the output are the origination constraints that lead to such a sound assignment. Indeed, the assignment of relies and guarantees is only sound under a particular set of origination constraints not shown. These constraints were part of the queries that were written by the protocol designer in order to produce the CPSA output that serves as input to this algorithm. A solution to this emerges as part of the second goal addressed in the next section of this report.

It is worthwhile to note that the algorithm filters out guarantees made by the strand being annotated with relies. Turning such filtering off makes it explicit that at each message reception, the strand still knows what it previously derived. Should the participants be completely stateless with respect to individual message transmissions and receptions, the principals would still have the information necessary to make branching decisions. Such a stateless, continuations-based implementation of a protocol participant may lessen the resource requirements of participating in multiple simultaneous executions of the protocol.

This algorithm's repetition of already observed guarantees clarifies that these guarantees can still be used on later nodes of the strand.

6 Origination Information

CPSA allows for cryptographic protocol designers to annotate transmission nodes with trust formulas and derive, using Algorithms 1 and 2 presented in the previous section, a sound assignment of trust formulas to receiving nodes under certain constraints. These constraints state that certain pieces of information are either uniquely originating or non-originating on the participating strands. Currently, this origination-constraint information is specified either as part of the protocol description or (as we saw in Listings 6 and 9) the skeleton used as a query to CPSA. As mentioned in the last section, this way of using CPSA does not make it clear how origination information can affect the branching behavior of a protocol. This section presents a method to introduce trust information into the language of the relies and guarantees to allow such information to interact with the branching-dependent trust formulas assigned to each node.

A representation of uniquely originating and non-originating information that views this information as something that can change over the time a protocol executes could be combined in sensible ways with trust systems such as the Public Key Infrastructure. This would be possible if the origination information was integrated into the rely-guarantee language. In that way, the constraint information would be free to be determined at run-time as opposed to statically at compile-time.

As an example, suppose partway through the execution of a protocol a participant learns from a certificate authority that a particular key has been compromised and it is now possible for an adversary to forge message signatures for the key's owner and decrypt messages destined to the key-holding principal. It is important for each principal involved in cryptographic exchanges with the compromised client to know whether or not it is safe to continue executing the protocol, branch to some other behavior, or abort altogether. If the principal knew the conditions under which each received message implied that a rely was uttered, that principal could determine if those conditions are met and if the rely can be safely combined with local trust information.

In this way, principals executing the protocol will not be confined to simply trusting some or all information from other principals. Instead, (with the help of the protocol designer) these principals will be able to know under what run-time conditions they may conclude guarantees made by those principals they do trust.

6.1 Algorithm Design

Origination information is used differently from trust information. Information uttered by a principal may be trusted or not depending on what the receiving principal knows about the transmitting principal. One might consider allowing principals to make a guarantee such as `key k is non-originating` and allow corresponding relies to look like `P says key k is non-originating`. However, unlike a guarantee `P_requests_resource_x` that has a self certifying rely `P says P_requests_resource_x`, the trust-information-containing rely formulas may be true or false due to external factors beyond the control of the principal involved in the transmission. Accordingly, it would not make sense to trust or not trust a principal's statements regarding the strands on which information originates.

Origination information behaves more like a contract, specifying under what preconditions certain other pieces of information become true. Therefore, the language of the relies and guarantees needs to be expanded to allow for this conditional nature.

Ideally, on the reception of a particular message, the principal could choose to accept or reject a set of conditions regarding the ingredients (such as nonces) and non-ingredients (keys) of a message and based on that understanding, conclude which bits of previously-uttered relies it would like to trust or not trust using the same local logic system outlined in the previous section.

6.2 Algorithm Implementation

The algorithm presented here takes as input a protocol definition annotated by the human protocol designer with trust formulas as guarantees (as before) and origination formulas as a predicate for a trivially true implication as explained shortly. The algorithm will produce output containing query skeletons for use in CPSA. The implication placed on the receiving nodes is of a specific form that does not allow the protocol designer to accidentally create an unsound rely-guarantee assignment (as not to regress from the first goal of this paper). Before processing, a guarantee on a node may take the form of `(implies (and (uniq` $\texttt{nonce}_1 \ldots \texttt{nonce}_n$ `) (non-orig` $\texttt{key}_1 \ldots \texttt{key}_m$ `)) true)`. Writing the origination information in this manner posits this information more like a question asking "Given this set of origination information is true, what else is true?". Before executing the algorithm, the trivially true value `true` is left as a placeholder on the right side of the implication. After using this algorithm in conjunction with CPSA and the rely-generation

algorithm, the `true` will be replaced by a stronger statement representing what relies must have been involved under those conditions to explain the node being annotated.

As a slight enhancement, we allow the origination constraints `uniq-orig` and `non-orig` in guarantees as well. Although this information is of no use to other participants for the reasons already stated, it represents that the principal believes these statements to be true before transmitting. The is purely for convenience; it allows the protocol designer to annotate the protocol in a more intuitive way. When the algorithm is generating queries, the constraint information found in the first half of the rely will be used along with any constraint information in guarantees that come previously on the same strand.

For simplicity of presentation, we now allow each node to be a tuple (strand, nodeNumber, annotation). The annotations are allowed to be the same context-free grammar as before with the addition of keywords `uniq-orig` and `non-orig` to represent uniquely originating and non-originating information respectively. These conjunctively-joined verbs are the *origination constraints* on that node. At the start, all transmission nodes are annotated and all receiving nodes have the trivially true statement consisting of a conjunction of one or more origination constraints implying *true*. Using Algorithm 3 to generate queries and a modified form of Algorithm 2 (shown as Algorithm 4) to process the output from CPSA, the right hand side of each implication will be filled with the guarantees that may be accepted if the key infrastructure or other mechanisms used by a principal can successfully validate the left hand side of the implication at the time the message is received.

Algorithm 3 generateQueries(list of strands *strands*) → list of skeletons

Let *answer* be a list of skeletons
for all *strand* in *strands* **do**
  for all *node* in *strand* **do**
    if *node* is a transmit node **then**
      Let *query* be a query skeleton for node *node*
      query.constraints =

$$(node.annotation.antecedent \wedge constraintsBefore(strand, node))$$

      answer.add(*query*)
    end if
  end for
end for
return *answer*

Algorithm 4 annotateNodes(list of nodes *nodes*, list of shapes *shapes*) → list of nodes

let *answer* be a list of annotated nodes
for all n in *nodes* **do**
$n.annotation =$

$$(n.annotation.antecedent) \rightarrow \left(\bigvee_{s \text{ realizing } n \in shapes} \left(\bigwedge_{g \in \text{ annotationsBefore}(n,s)} g \right) \right)$$

$answer.add(n)$
end for
return *answer*

6.3 Algorithm Analysis

6.3.1 Termination

Algorithm 3 terminates as it loops over every node of all strands exactly once. Algorithm 4 terminates as it calls $annotationsBefore$ exactly once for each node being annotated. The $annotationsBefore$ function is expected to return the annotations found on each transmission node on the strand before the current node. Since strands are finite this function will always terminate.

6.3.2 Soundness

Algorithm 3 generates one query for every node. This query has as its constraints the constraint-half (the first half) of the user supplied implication of the node joined conjunctively with any constraints found on transmission nodes occurring before the node. The algorithm is sound providing we specify that guarantees remain true over the time a protocol executes.

Algorithm 4 is sound for the exact reasoning that Algorithm 2 is sound, with the addition that the rely is placed in the second half of the implication, indicating its veracity is dependent on the origination constraints found in the implication.

6.4 Algorithm Architecture

This algorithm was implemented using Python and expanded on the work of the previous goal. The low-level S-expression parsing script by Elliott Franco Drabek along with the more high-level CPSA parsing script previously developed has been adapted to also be used by the script implementing the below algorithm. A copy of this script and an explanation of its function can be found in Appendix B.

6.5 Example Execution

Let us consider a protocol where clients may request market data from a provider. Home clients use this service for free and as a consequence receive market data back that is approximately correct. Corporate clients have a paid account with the provider and as a result receive more accurate data back from the queries made providing that their account still exists.

In this protocol, the client (home or corporate) first agrees with the server on a shared key with which to continue execution of the protocol. The client will then make the request encrypted with this key. The messages from a corporate client and a home client have identical ingredients. The server decides to send an approximate data value or an exact data value based on the presence of an account for the value bound to the name identifier of previous messages.

The branching behavior of the server is represented by separate strands, one for each of the two branches possible. If the server determines (by querying some local data store) that the client making the request is a home user, it can reply with a message containing ingredients that reflect the fact that it is destined for a home user. This is represented by the string literal "approx_data_is". Otherwise, if the local data store indicates that the requesting client is a corporate subscriber, the data ingredient is reported along with the "data_is" literal. The annotations attached to the data value transmitted by the server indicate the assertion that the client's identity has been determined and that the data value being returned is either exactly or approximately equal to the value found in the message.

In order to explore behavior compatible with particular origination constraint information, the protocol designer might wish to annotate the protocol as in Listing 15, placing an empty trust implication on each receive.

Listing 15: Stock server protocol before processing

```
(defprotocol branch1 basic
  (defrole corp_client
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (send (enc n_a a d (pubk b)))
      (recv (enc n_a k b (pubk a)))
      (send (enc k (pubk b)))
      (recv (enc "data_is" v k)))
    (annotations a
      (0 (requests b d n_a))
      (1 (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a)) true))
      (3 (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a k)) true))
    ))
  (defrole home_client
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (send (enc n_a a d (pubk b)))
      (recv (enc n_a k b (pubk a)))
      (send (enc k (pubk b)))
      (recv (enc "approx_data_is" v k)))
    (annotations a
      (0 (requests b d n_a))
      (1 (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a)) true))
      (3 (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a k)) true))
    ))
  (defrole server_corp_branch
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (recv (enc n_a a d (pubk b)))
      (send (enc n_a k b (pubk a)))
      (recv (enc k (pubk b)))
      (send (enc "data_is" v k)))
    (annotations b
      (2 (implies (and (non-orig (privk a)) (uniq-orig k)) true))
      (3 (and (corp_subscriber a d) (curr_val d v n_a)))
    ))
  (defrole server_home_branch
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (recv (enc n_a a d (pubk b)))
      (send (enc n_a k b (pubk a)))
      (recv (enc k (pubk b)))
      (send (enc "approx_data_is" v k)))
    (annotations b
      (2 (implies (and (non-orig (privk a)) (uniq-orig k)) true))
      (3 (and (home_subscriber a d) (approx_val d v n_a))))))
```

After processing Listing 15 with both algorithms, the output shows each implication has been filled in with the strongest statement possible. This can be seen in Listing 16.

Listing 16: Stock server protocol after processing

```
(defprotocol branch1 basic
  (defrole corp_client
    (a b name)
    (n_a d v text)
    (k skey)
    (trace (send (enc n_a a d (pubk b))) (recv (enc n_a k b (pubk a)))
      (send (enc k (pubk b))) (recv (enc "data_is" v k)))
    (annotations a (0 (requests b d n_a))
      (1
        (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a))
          true))
      (3
        (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a k))
          (says b (and (corp_subscriber a d) (curr_val d v n_a)))))))
  (defrole home_client
    (a b name)
    (n_a d v text)
    (k skey)
    (trace (send (enc n_a a d (pubk b))) (recv (enc n_a k b (pubk a)))
      (send (enc k (pubk b))) (recv (enc "approx_data_is" v k)))
    (annotations a (0 (requests b d n_a))
      (1
        (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a))
          true))
      (3
        (implies (and (non-orig (privk a) (privk b)) (uniq-orig n_a k))
          (says b (and (home_subscriber a d) (approx_val d v n_a)))))))
  (defrole server_corp_branch
    (a b name)
    (n_a d v text)
    (k skey)
    (trace (recv (enc n_a a d (pubk b))) (send (enc n_a k b (pubk a)))
      (recv (enc k (pubk b))) (send (enc "data_is" v k)))
    (annotations b (0 true)
      (2
        (implies (and (non-orig (privk a)) (uniq-orig k))
          (says a (requests b d n_a))))
      (3 (and (corp_subscriber a d) (curr_val d v n_a)))))
  (defrole server_home_branch
    (a b name)
    (n_a d v text)
    (k skey)
    (trace (recv (enc n_a a d (pubk b))) (send (enc n_a k b (pubk a)))
      (recv (enc k (pubk b))) (send (enc "approx_data_is" v k)))
    (annotations b (0 true)
      (2
        (implies (and (non-orig (privk a)) (uniq-orig k))
          (says a (requests b d n_a))))
      (3 (and (home_subscriber a d) (approx_val d v n_a))))))
```

The implications on the last node of both the *home_client* and *corp_client* reflect that the participants can conclude that the corresponding guarantees were uttered by the server as long as the origination information in the first half of the implication can be verified. Assuming that the clients trust the server, each client would be able to conclude that they were receiving the information they expected to receive. In particular, the corporate subscriber would be certain that the server was aware that the requesting participant was a corporate subscriber and that this participant was getting the accurate information that presumable the subscriber is paying for.

6.6 Discussion

As with the rely-generation algorithm in the previous section, the work presented in this section benefits both the human designing the protocol and the implementation of each participant in the protocol. The protocol designer can add reasonable constraints to different nodes in order to explore the resulting behavior without having to write each query manually.

For the protocol participant, the algorithm presented in the paper makes clear the exact requirements necessary to draw a conclusion based on a particular message reception. The semantics make it clear how relies could be integrated with a key distribution/revocation system such as the Public Key Infrastructure. Furthermore, the implications in the rely-guarantee language allows CPSA to agree semantically with the syntax of CPPL, which specifies the preconditions and post conditions of cryptographic routines. Integrating CPSA and CPPL will allow for a more unified approach to protocol design, analysis, and implementation.

7 Further Enhancements

As an improvement on Algorithm 3, this project investigated a combinatorial approach to determining possible shapes that can be produced by a particular protocol. In this modification, the protocol designer does not add any origination constraints to the protocol. Instead of generating a single skeleton query for each reception node in the protocol, we instead generate each possible query combination for each reception node.

7.1 Algorithm Implementation

The algorithm presented here takes as input a protocol definition annotated by the human protocol designer with only trust formulas assigned as guarantees. The algorithm produces as output a set of skeletons queries for use in CPSA. Instead of producing one query for each reception node, each possible query is generated from a set of possible constraints. The set of possible constraints considers the inverse of each key used as a candidate for inclusions as non-originating in the query and each message ingredient as candidate for uniquely originating in the query.

7.2 Example Execution

Returning to the quote server of Example 15, suppose we start instead with simply the protocol annotated with rely formulas similar to what would be used as input to the original rely-generation algorithm presented in Section 5.

Listing 17: Stock server protocol before processing

```
(defprotocol branch1 basic
  (defrole corp_client
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (send (enc n_a a d (pubk b)))
      (recv (enc n_a k b (pubk a)))
      (send (enc k (pubk b)))
      (recv (enc "data_is" v k)))
    (annotations a
      (0 (requests b d n_a))
    ))
  (defrole home_client
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (send (enc n_a a d (pubk b)))
      (recv (enc n_a k b (pubk a)))
      (send (enc k (pubk b)))
      (recv (enc "approx_data_is" v k)))
    (annotations a
      (0 (requests b d n_a))
    ))
  (defrole server_corp_branch
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (recv (enc n_a a d (pubk b)))
      (send (enc n_a k b (pubk a)))
      (recv (enc k (pubk b)))
      (send (enc "data_is" v k)))
    (annotations b
      (3 (and (corp_subscriber a d) (curr_val d v n_a)))
    ))
  (defrole server_home_branch
    (vars (a b name) (n_a d v text) (k skey))
    (trace
      (recv (enc n_a a d (pubk b)))
      (send (enc n_a k b (pubk a)))
      (recv (enc k (pubk b)))
      (send (enc "approx_data_is" v k)))
    (annotations b
      (3 (and (home_subscriber a d) (approx_val d v n_a))))))
```

Using the modified scripts to brute force all combinations of constraints, annotations are emitted with implications that resulted in interesting behavior. The annotation generated for the final receive node of the corporate client is shown in Listing 18

Listing 18: Annotation generated for final receive node of corporate subscriber

```
(or
  (implies
    ((non-orig (privk a) (privk b)) (uniq-orig n_a d a k))
    (says b (and (corp_subscriber a d) (curr_val d v n_a))))
  (implies ((non-orig (privk a) (privk b)) (uniq-orig n_a b k))
    (says b (and (corp_subscriber a d) (curr_val d v n_a))))
  (implies
    ((non-orig (privk a) (privk b)) (uniq-orig n_a a b k))
    (says b (and (corp_subscriber a d) (curr_val d v n_a))))
  (implies ((non-orig (privk a) (privk b)) (uniq-orig n_a a k))
    (says b (and (corp_subscriber a d) (curr_val d v n_a))))
  (implies
    ((non-orig (privk a) (privk b)) (uniq-orig n_a d b k))
    (says b (and (corp_subscriber a d) (curr_val d v n_a))))
  (implies ((non-orig (privk a) (privk b)) (uniq-orig n_a k))
    (says b (and (corp_subscriber a d) (curr_val d v n_a))))
  (implies ((non-orig (privk a) (privk b)) (uniq-orig n_a d k))
    (says b (and (corp_subscriber a d) (curr_val d v n_a))))))))
```

The output shows that seven different combinations of origination constraints produced a shape involving some guarantee activity. In this case, each interesting result produced the same shape consisting of the server determining a corporate subscriber was making a request and returning the current values. Inspecting these results we see that the least constrained value is the sixth implication, whose antecedent is exactly the one explored in the previous section. All other implications produced had slightly more constrained behavior and resulted in the same rely assignment.

This example shows that a combinatorial approach to analyzing protocols can elucidate how ingredients modify the behavior of a protocol's execution. In this example, the uncompromised keys and non-origination of both the nonce n_a and the shared key k were critical in the corporate client being able to conclude that the server had established its identity correctly as a paying customer.

7.3 Discussion

This combinatorial approach to determining sufficient constraints necessary to generate certain guarantees will further aid the protocol designer in determining how individual message contents affect the shapes that can be generated. Such a tool could be useful for both a protocol designer working on a new protocol and a student attempting to gain an understanding on how an existing protocol works.

8 Conclusions

The method for automated rely-generation presented in this paper has been shown to generate a fully annotated protocol from a protocol decorated with guarantees only. Not only should this algorithm reduce the amount of effort required on behalf of the protocol designer to make sound rely-guarantee assignments, the output of the algorithm should aid in the implementation of the protocol. The output presents precisely the information each participant is allowed to conclude on each message reception.

By using implications to introduce the trust constraints of skeleton queries into the language of the relies and guarantees, the implementation requirements for each principal are made explicit. The output shows exactly which propositions need to be confirmed or assumed in order for a principal to conclude anything useful about the other participants based solely on the form of the ingredients in the received message. This improvement more tightly couples the interaction between the (CPSA-proven) protocol behavior and the trust management logic contained in each principal's implementation.

The integration of the solutions for the two above goals serve to both increase the clarity of the output and decrease the work required from the protocol designer in automatically generating relies. The ambiguity introduced in the output of the automatically generated relies not hinting to the assumptions for which they hold is solved by moving the constraint information into the relies and guarantees. As the origination-information script also automatically generates the query skeletons to be used, the protocol designer needs only to annotate guarantees and make sane assumptions on origination information in either the relies or guarantees.

The tools developed in this paper should also serve well the student studying cryptographic protocols. In particular, the combinatorial approach to constraint information explored towards the end of this project shines light on the exact manner in which ingredients affect protocol behavior in a way that is clear and concise. Such a tool could be useful in teaching protocol design.

8.1 Future Work

As CPPL is concerned with compiling protocol descriptions into working implementations of the protocol, a valuable future project would include a CPPL-to-CPSA converter. Such a converter would allow a protocol designer to write their protocol definition in CPPL. This definition could be compiled down to CPSA for analysis by using the tool before

processing with CPSA and the scripts presented in this paper. After any number of revisions necessary to elicit the desired behavior from the analysis, the protocol designer could simply compile the CPPL protocol description down to an implementation. The constraint-containing implications of the relies would mesh well with the preconditions specified as part of routines in CPPL.

As it stands, the use of standard input and output for passing information between CPSA and the scripts developed in this project results in these programs performing multiple parsing operations on the same data. If performance proves to be a concern in the future, the logic of the solutions outlined in this paper could be integrated with CPSA. In testing, the performance decrease was not so large as to be any bother.

It would be worth investigating how the combinatorial approach to constraint generation could be improved. As it stands, the output only states the conclusions that can be drawn from each combination of constraint information. If this can be automated, perhaps the need and placement of nonces can be automated as well in order to reach goals specified by the protocol designer.

References

[1] Shaddin Doghmi, Joshua Guttman, and F. Thayer. Searching for shapes in cryptographic protocols. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 523–537. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-71209-1_41.

[2] Joshua Guttman. Security theorems via model theory. In *EXPRESS: Expressive-ness in Concurrency (EPTCS)*. 2009. doi:10.4204/EPTCS.8.5.

[3] Joshua Guttman, Jonathan Herzog, John Ramsdell, and Brian Sniffen. Programming cryptographic protocols. In Rocco De Nicola and Davide Sangiorgi, editors, *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 116–145. Springer Berlin / Heidelberg, 2005. 10.1007/11580850_8.

[4] Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In David Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 325–339. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24725-8_23.

[5] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

A Rely Generation Code

A.1 Source

The first script developed by Elliott Franco Drabek is used for the low level parsing of the S-expression language used as both the input and output language of CPSA.

Listing 19: S-expression parsing script, written by Elliott Franco Drabek: s_expr_gen.py

```
##############################################################################
# Super-primitive Python s-expression parser by Elliott Franco Drabek
#
# This might save you the two hours it took me to put it together, test it, and
# (moderately) speed it up.
#
# It only understands lists and atoms.  It understands atoms to be consecutive
# runs of non-whitespace, non-parenthesis characters.  It converts all-digit
# atoms into ints.
#
# Public domain
#
# No guarantees
#
# Last modified Fri Aug 29 13:57:25 EDT 2003

import sys

##############################################################################

def _gen_tokens(file):
  for line in file:
    line_len = len(line)
    left = 0

    while left < line_len:
      c = line[left]

      if c.isspace():
        left += 1
      elif c in '()':
        yield c
        left += 1

      else:
        right = left + 1
        while right < line_len:
          c = line[right]
          if c.isspace() or c in '()':
            break

          right += 1

        token = line[left:right]
        if token.isdigit():
          token = int(token)
        yield token

        left = right

def read_all(file=sys.stdin):
  stack = []
```

```
  for token in _gen_tokens(file):
    if token == '(':
      stack.append([])

    elif token == ')':
      top = stack.pop()
      if len(stack) == 0:
        yield top
      else:
        stack[-1].append(top)

    else:
      stack[-1].append(token)

  assert len(stack) == 0
```

One script is responsible for parsing the input expressions into the internal representation of the protocol and shapes. This script can serve as the base for future work with CPSA in Python.

Listing 20: CPSA language parsing script: parse.py

```
#! /usr/bin/env python

from types import *
import s_expr_gen

def s_expr_to_str(e):
  if type(e) is ListType:
    return '(%s)' % ' '.join(map(s_expr_to_str, e))
  else:
    return str(e)
protocol = None

class Precedes:
  def __init__(self, s):
    self.list = []
    for sub in s:
      if type(sub) is ListType:
        self.list.append(sub)

class Skeleton:
  def __init__(self, s, p):
    self.protocol = p
    self.precedes = None
    self.is_shape = False
    self.constraints = []
    self.def_strands = []
    self.query_role_name = "" #the first defstrand
    self.query_role_node = -1 #the first defstrand
    for sub in s:
      if sub[0] == "precedes":
        self.precedes = Precedes(sub)
      if sub[0] == "shape":
        self.is_shape = True
      if sub[0] == "uniq-orig" or sub[0] == "non-orig":
        self.constraints.append(sub)
      if sub[0] == "defstrand":
        #the strand indices are the order in which we read them
        #NOTE WELL: the nodes are 1 based!
        self.def_strands.append(sub[1])
        if self.query_role_name == '' and self.query_role_node == -1:
```

```
            self.query_role_name = sub[1]
            self.query_role_node = sub[2] - 1
    def get_query_node(self):
      #this functio nreturns the query that resulted in this shape
      return (self.query_role_name, self.query_role_node)
    def get_precedes(self):
      #this function returns a list like (((resp, 0),(resp2 1)), ((),()), ((),()))
      list = []
      if (self.precedes != None and len(self.precedes.list) > 0):
        for ((s1, n1),(s2, n2)) in self.precedes.list:
          list.append(((self.def_strands[s1], n1),(self.def_strands[s2], n2)))
      return list
    def get_all_guarentees_before(self, role_name, node_index):
      precedes_line = self.get_precedes()
      answers = {}
      in_progress = []
      in_progress.append((role_name, node_index))
      while len(in_progress) > 0:
        (name,node) = in_progress.pop(0)
        #if this is a node of interest (send) and has a non-trivial annotation, add it
        if ((self.protocol.get_role_by_name(name).trace.messages[node].type == 'send') and
            (node in self.protocol.get_role_by_name(name).annotations) and
            (self.protocol.get_role_by_name(name).annotations[node] != "(true)")):
          role = self.protocol.get_role_by_name(name)
          answers[(name,node)] = ["says", role.annotation_label, role.annotations[node]]
        #go through the precedes line and add any that precede this node
        for ((name1, node1), (name2, node2)) in precedes_line:
          if name2 == name and node2 == node:
            in_progress.append((name1, node1))

        #if this is not the first node on this role, put the previous one on
        if node > 0:
          in_progress.append((name, node - 1))
      return answers.values()


class Message:
  def __init__(self, s):
    self.type = s[0]
    self.value = s[1]
  def to_string(self):
    return "(%s %s)" % (self.type, s_expr_to_str(self.value))

class Trace:
  def __init__(self, s):
    self.messages = []
    for sub in s:
      if sub[0] == "send" or sub[0] == "recv":
        self.messages.append(Message(sub))
  def to_string(self):
   messages_string = ''
   for message in self.messages:
     messages_string += message.to_string()
   return "(trace %s)" % messages_string

class Role:
  def __init__(self, s):
    self.name = s[1]
    self.trace = None
    self.vars_string = ''
    self.annotation_label = ''
    self.annotations = {}
    for sub in s:
      if sub[0] == "vars":
        self.vars_string = s_expr_to_str(sub)[6:-1]
```

```
      if sub[0] == "trace":
        self.trace = Trace(sub)
      if sub[0] == "annotations":
        self.annotation_label = sub[1]
        for sub2 in sub:
          if type(sub2) is ListType:
            self.annotations[sub2[0]] = sub2[1]
  #this function returns all the constraints uttered on
  # message sends by this strand before node n. They must
  # be true (for the same strand) and therefor can be
  # used for shape generation
  def get_constraints_before(self, n):
    constraints = []
    for (message_index, message) in enumerate(self.trace.messages):
      if message_index < n and message.type == "send" and message_index in self.annotations:
        #extract any origination infomation from this node's annotation
        constraints += extract_constraints_from_guarantee(self.annotations[message_index])
    return constraints


  def to_string(self):
    trace_string = self.trace.to_string()
    annotations_string = ''
    for node_index in self.annotations.keys():
      annotations_string += "(%d %s)" % \
        (node_index, s_expr_to_str(self.annotations[node_index]))
    if len(annotations_string):
      annotations_string = "(annotations %s %s)" % \
        (self.annotation_label, annotations_string)
    return "(defrole %s %s %s %s)" % \
      (self.name, self.vars_string, trace_string, annotations_string)



class Protocol:
  def __init__(self, s):
    self.name = s[1]
    self.grammar = s[2]
    self.roles = []
    self.shapes = []

    if self.grammar == 'basic':
      for sub in s:
        if sub[0] == 'defrole':
          self.roles.append(Role(sub))
    else:
      print "Erorr!"

  def get_role(self, shape_index, strand_index):
    #each shape uses different indices
    role_name = self.shapes[shape_index].def_strands[strand_index]
    for role in self.roles:
      if role_name == role.name:
        return role
    return None

  def get_role_by_name(self, name):
    for role in self.roles:
      if role.name == name:
        return role
    return None

  def to_string(self):
    roles_string = ''
    for role in self.roles:
```

```
      roles_string += role.to_string()
    string = "(defprotocol %s %s %s)" % (self.name, self.grammar, roles_string)
    return string

############ Utility functions ##################################################

#takes a parsed s-expression and simplifies it
# (and)          goes to true
# (and expr)     goes to expr
# (or)           goes to false
# (or expr)      goes to expr
def simplify(e): #an expression, may be a list or a single item
  if type(e) is ListType:
    length = len(e)
    if e[0] == "and":
      if length == 1:
        return "true"
      elif length == 2:
        return simplify(e[1])
      else:
        return map(simplify, e)
    elif e[0] == "or":
      if length == 1:
        return "false"
      elif length == 2:
        return simplify(e[1])
      else:
        return map(simplify, e)
    else:
      return map(simplify, e)
  else:
    return e

#take a guarantee annotation and extracts origination constraints
def extract_constraints_from_rely(annotation):
  #NOTE - assumes that if an implication is found
  # on a rely it is of the form "constraints -> True"
  if annotation[0] == "implies":
    annotation = annotation[1]
    if annotation[0] == "and":
      return annotation[1:]
    else:
      return annotation
  return ""

def extract_constraints_from_guarantee(annotation):
  #NOTE - assumes that a guarantee is
  # either an atom or conjunction of atoms
  constraints = []
  if annotation[0] == "uniq-orig" or annotation[0] == "non-orig":
    constraints.append(annotation)
  elif annotation[0] == 'and':
    for sub in annotation:
      constraints += extract_constraints_from_guarantee(sub)
  return constraints
```

Lastly, the rely generation itself is performed in a third script that utilizes the data structures created by the parsing code to carry out the high-level tasks of the algorithm implementation.

Listing 21: Rely generation script: rely_generation.py

```
#! /usr/bin/env python

from types import *
import s_expr_gen
from parse import *

def input_to_protocol():
  protocol = None
  #parse the input
  for s in s_expr_gen.read_all():
    if not protocol and s[0] == 'defprotocol':
      protocol = Protocol(s)
    if protocol:
      if s[0] == "defskeleton":
        skeleton = Skeleton(s, protocol)
        if skeleton.is_shape:
          protocol.shapes.append(skeleton)
  return protocol

def main():
  protocol = input_to_protocol()

  if protocol:
    #gather up a list of receive nodes that need relies
    #we may or may not have queries that give these
    #given nodes
    recieve_nodes = []
    for (strand_index, role) in enumerate(protocol.roles):
      for (node_index, message) in enumerate(role.trace.messages):
        if message.type == "recv":
          recieve_nodes.append((protocol.roles[strand_index].name,node_index))

    for (role_name, node_index) in recieve_nodes:
      disjunction = ['or']
      for (shape_index, shape) in enumerate(protocol.shapes):
        if shape.get_query_node() == (role_name, node_index):
          conjunction = ['and']
          guarentees = shape.get_all_guarentees_before(role_name, node_index)
          for g in guarentees:
            #don't add things this strand already knows about itself
            if not (g[0] == "says" and
                    g[1] == protocol.get_role_by_name(role_name).annotation_label):
              conjunction.append(g)
          disjunction.append(conjunction)
      #we now have the disjunction for this node/index
      if len(disjunction) > 1: #dont add an empty one
        disjunction = simplify(disjunction)

        if node_index in protocol.get_role_by_name(role_name).annotations:
          annotation = protocol.get_role_by_name(role_name).annotations[node_index]
          if annotation[0] == "implies" and annotation[2] == "true":
            annotation[2] = disjunction
            protocol.get_role_by_name(role_name).annotations[node_index] = annotation
          else:
            #NOTE - blowing over the value that was there
            protocol.get_role_by_name(role_name).annotations[node_index] = disjunction
        else:
          protocol.get_role_by_name(role_name).annotations[node_index] = disjunction


  print protocol.to_string()

main()
```

A.2 Usage

Suppose we have a protocol definition and some skeletons querying some receive nodes in a file example.scm. We can use CPSA in combination with the rely-generation script to produce an annotated protocol as shown in Listing 22

Listing 22: Calling the rel-generation script

```
$cpsa example.scm | ./rely-generation.py | cpsapp
```

B Origination Code

B.1 Source

In addition to the code in Listings 19 and 20, the origination information algorithm is implemented in the following code listing.

Listing 23: Origination information algorithm: origination.py

```
#! /usr/bin/env python

from types import *
import s_expr_gen
from parse import *

def input_to_protocol():
  protocol = None
  #parse the input
  for s in s_expr_gen.read_all():
    if not protocol and s[0] == 'defprotocol':
      protocol = Protocol(s)
  return protocol

def main():
  protocol = input_to_protocol()

  if protocol:
    #enumerate over transmit nodes
    for (strand_index, role) in enumerate(protocol.roles):
      for (node_index, message) in enumerate(role.trace.messages):
        if message.type == "recv":

          constraints = []
          #add constraints placed in an implication on this receive
          if node_index in protocol.roles[strand_index].annotations:
            constraints = extract_constraints_from_rely(
              protocol.roles[strand_index].annotations[node_index])
          #add any constraints placed on sends of this strand before this node
          constraints += protocol.roles[strand_index].get_constraints_before(node_index)

          #generate each defskeleton
          print "(defskeleton␣%s␣(vars␣%s)" % \
            (protocol.name, protocol.roles[strand_index].vars_string)
          print "(defstrand␣%s␣%s␣%s)" % \
            (protocol.roles[strand_index].name, node_index + 1,
            generate_mappings(protocol.roles[strand_index].vars_string))
          print "%s" % s_expr_to_str(constraints)[1:-1]
          print ")\n"

############ helper functions ##################################

#take a string of variables like (cat dog) and produces (cat cat) (dog dog)
def generate_mappings(var_string):
  ret = ""
  for word in var_string.split("␣"):
    if word[-1] != ")": #the last word of each set describes the set (key, text, etc)
      if word[0] == "(":
        word = word[1:]
      ret += "(%s␣%s)␣" % (word, word)
  return ret
```

```
#take a guarantee annotation and extracts origination constraints
def extract_constraints_from_rely(annotation):
  #NOTE - assumes that an implication on a rely is of the form "constraints -> True"
  if annotation[0] == "implies":
    annotation = annotation[1]
    if annotation[0] == "and":
      return annotation[1:]
    else:
      return annotation
  return ""


main()
```

B.2 Usage

Suppose we have an input file example.scm that has implications as the annotations on some receive nodes that look like `(implies [constraint information] true)`. We would like to generate queries to be used in CPSA. We can do this with the orignation script as shown in Listing 24.

Listing 24: Calling the origiantion script

```
$cat example.scm | ./origination.py
```

We can recombine these questions with the original protocol definition and use CPSA along with the same rely-generation script as before to annotate the protocol as shown in Listing 25.

Listing 25: Using the origiantion script in conjunction with the rely-generation script

```
$cat examples.scm | ./origination.py | cat examples.scm - | cpsa |
./rely-generation.py | cpsapp
```

C Combinatorial Approach

C.1 Source

The combinatorial approach is implemented utilizing two scripts shown in Listings 26 and 27.

Listing 26: Origination information combinatorial approach script 1: origination-try-all.py

```
#! /usr/bin/env python

from types import *
import s_expr_gen
from parse import *

def input_to_protocol():
  protocol = None
  #parse the input
  for s in s_expr_gen.read_all():
    if not protocol and s[0] == 'defprotocol':
      protocol = Protocol(s)
  return protocol

def main():
  protocol = input_to_protocol()

  if protocol:
    #enumerate over transmit nodes
    for (strand_index, role) in enumerate(protocol.roles):
      for (node_index, message) in enumerate(role.trace.messages):
        if message.type == "recv":


          possible_constraints = {} #using a dictionary to avoid duplicates
          for i in range(0,node_index):
            message_contents = protocol.roles[strand_index].trace.messages[i].value
            for constraint in generate_possible_key_constraints(message_contents):
              possible_constraints[s_expr_to_str(constraint)] = 1
            for constraint in generate_possible_ingredient_constraints(message_contents):
              possible_constraints[s_expr_to_str(constraint)] = 1

          possible_constraints = possible_constraints.keys()

          #they are different combinations of constraint possibilities
          #each possible constraint can be included or excluded
          sets = 2**len(possible_constraints)
          for bit_mask in range(0, sets-1):
            constraints = generate_combination(possible_constraints, bit_mask)
            #NOTE we are ignoring previous constraints on this strand....
            #this makes sense for a combinatorial approach
            print "(defskeleton %s (vars %s)" % \
              (protocol.name, protocol.roles[strand_index].vars_string)
            print "(defstrand %s %s %s)" % \
              (protocol.roles[strand_index].name, node_index + 1, \
              generate_mappings(protocol.roles[strand_index].vars_string))
            print "%s" % s_expr_to_str(constraints)[1:-1]
            print ")\n"

############ helper functions ##################################

def generate_possible_key_constraints(message):
  constraints = []
```

```
  keys = extract_keys(message)
  for key in keys:
    constraints.append(['non-orig', key])
  return constraints

def extract_keys(message):
  #supports extracting keys from the "enc" expression
  keys = []
  if type(message) is ListType and message[0] == "enc":
    key = message[-1]
    if type(key) is ListType and (key[0] == "pubk" or key[0] == "privk"):
      key[0] = "privk" #set to the decryption key
      keys.append(key)
  for sub in message[1:-1]:
    keys += extract_keys(sub)
  return keys


def generate_possible_ingredient_constraints(message):
  constraints = []
  ingredients = extract_ingredients(message)
  for ing in ingredients:
    constraints.append(['uniq-orig',ing])
  return constraints

def extract_ingredients(message):
  ingredients = []
  if type(message) is ListType:
    if message[0] == "enc":
      message = message[1:-1] #lose the "enc" amd the key
    if message[0] == "cat":
      message = message[1:] # lose the cat keyword
    for sub in message:
      ingredients += extract_ingredients(sub)
  else:
    if message[0] != '"': #dont include string literals
      ingredients.append(message)
  return ingredients

def generate_combination(choices, bitmask):
  ret = []
  position = 0
  while (bitmask != 0):
    if (bitmask & 1):
      ret.append(choices[position])
    bitmask >>= 1
    position += 1
  return ret

#take a string of variables like (cat dog) and produces (cat cat) (dog dog)
def generate_mappings(var_string):
  ret = ""
  for word in var_string.split(" "):
    if word[-1] != ")": #the last word of each set describes the set (key, text, etc)
      if word[0] == "(":
        word = word[1:]
      ret += "(%s %s) " % (word, word)
  return ret

main()
```

Listing 27: Origination information combinatorial approach script 2: rely-generation-all.py

```
#! /usr/bin/env python
```

```
from types import *
import s_expr_gen
from parse import *

def input_to_protocol():
  protocol = None
  #parse the input
  for s in s_expr_gen.read_all():
    if not protocol and s[0] == 'defprotocol':
      protocol = Protocol(s)
    if protocol:
      if s[0] == "defskeleton":
        skeleton = Skeleton(s, protocol)
        if skeleton.is_shape:
          protocol.shapes.append(skeleton)
  return protocol

def main():
  protocol = input_to_protocol()

  if protocol:
    #gather up a list of receive nodes that need relies
    #we may or may not have queries that give these
    #given nodes
    recieve_nodes = []
    for (strand_index, role) in enumerate(protocol.roles):
      for (node_index, message) in enumerate(role.trace.messages):
        if message.type == "recv":
          recieve_nodes.append((protocol.roles[strand_index].name,node_index))

    for (role_name, node_index) in recieve_nodes:
      scenarios = {} #each scenario is what results from a particular set of constraints for th
      for (shape_index, shape) in enumerate(protocol.shapes):
        if shape.get_query_node() == (role_name, node_index):
          constraints = s_expr_to_str(shape.constraints)
          conjunction = ['and']
          guarentees = shape.get_all_guarentees_before(role_name, node_index)
          for g in guarentees:
            if not (g[0] == "says" and g[1] == protocol.get_role_by_name(role_name).annotation_
              conjunction.append(g)
          if constraints in scenarios:
            scenarios[constraints].append(conjunction)
          else:
            scenarios[constraints] = [conjunction]

      #we now have the all the disjunctions for this node/index stored as scenarios
      annotation = ['or']
      for scenario in scenarios:
        disjunction = ['or']
        for disjunct in scenarios[scenario]:
          if len(disjunct) > 1:
            disjunction.append(disjunct)
        if len(disjunction) > 1:
          annotation.append(["implies", scenario, disjunction])
      if len(annotation) > 1:
        protocol.get_role_by_name(role_name).annotations[node_index] = annotation


  print protocol.to_string()

main()
```

C.2 Usage

The first script is used to generate the queries and the second script is used to condense the output of CPSA to form the relies. Listing 28 shows the combined usage of the two scripts to annotate a protocol in the file example.scm.

Listing 28: Calling the origiantion script

```
$cat example.scm | ./origination-try-all.py | cat example.scm - |
cpsa | ./rely-generation-all.py | cpsapp
```